4. Basic building blocks of concurrency

Nelma Moreira & **José Proença**

Concurrent programming (CC3040) 2023/2024

CISTER – U.Porto, Porto, Portugal

`https://fm-dcc.github.io/pc2324`

Overview

We are here

Blocks of sequential code running concurrently and sharing memory:

- What is Scala?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- Futures and promises
- Actor model (maybe)

What we will see

- Tread pools: Executor and ExecutionContext
- Non-blocking synchronisation – compare-and-swap (CAS)
- Lazy (concurrent) values
- Concurrent collections
- Running OS processes

Existing thread pools in Scala

Executor interface

```
Executor executor =
    $\textit{anExecutor}$;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

```
import scala.concurrent._
import java.util.concurrent.ForkJoinPool

object ExecutorsCreate extends App {
  val executor = new ForkJoinPool
  executor.execute(new Runnable {
    def run() = log("This␣task␣is␣run␣
        asynchronously.")
  })
  Thread.sleep(500) // not needed with
      fork:=false in SBT
}
```

- Executor: can start a new thread, an existing one, or the current one
- Abstracts from the management of threads
- ExecutorService: API that extends Executor with `shutdown`
  - `executor.shutdown` $\longrightarrow$ executes all tasks and then stops working threads
  - `executor.awaitTermination(...)` $\longrightarrow$ force termination if, after a given time, the tasks are not completed

Scala's ExecutionContext

```
import scala.concurrent._
object ExecutionContextGlobal extends App {
  val ectx = ExecutionContext.global
  ectx.execute(new Runnable {
    def run() = log("Running␣on␣the␣execution␣context.")
  })
  Thread.sleep(500)
}
```

```
object ExecutionContextCreate extends App {
  val pool = new forkjoin.ForkJoinPool(2)
  val ectx = ExecutionContext.fromExecutorService(pool)
  ectx.execute(new Runnable {
    def run() = log("Running␣on␣the␣execution␣context␣
          again.")
  })
  Thread.sleep(500)
}
```

- `scala.concurrent`: has `ExecutionContext`
- Similar to `Executor` but more Scala specific
- often the `ExecutionContext` is used as implicit parameter
- `global`: default execution context (internally uses a `ForkJoinPool`)
- `fromExecutorService`: creates `ExecutionContext` from `ExecutorService`

Simplifying the execution

Similar to threads:

```
def thread(body: =>Unit): Thread
    = {
  val t = new Thread {
    override def run() = body
  }
  t.start()
  t
}
```

We now define `execute`

```
def execute(body: =>Unit) =
    ExecutionContext.global.execute(
     new Runnable { def run() = body }
)
// For example:
object ExecutionContextSleep extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

Avoid blocking indefinitely

```
object ExecutionContextSleep
       extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

- Expected: all executions terminate after 2s
- Result: only some execute after 2s

Avoid blocking indefinitely

```
object ExecutionContextSleep
       extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task␣$i␣completed.")
  }
  Thread.sleep(10000)
}
```

- Expected: all executions terminate after 2s
- Result: only some execute after 2s

- Using quad-core CPU with hyper threading
- `global` has 8 threads in the thread pool

Avoid blocking indefinitely

```
object ExecutionContextSleep
       extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task $i completed.")
  }
  Thread.sleep(10000)
}
```

- Expected: all executions terminate after 2s
- Result: only some execute after 2s

- Using quad-core CPU with hyper threading
- `global` has 8 threads in the thread pool

- executes tasks in batches of 8
- after 2s, 8 tasks print "completed"
- after 2s more, 8 more print "completed"
- sleep: all enter a timed waiting state

Avoid blocking indefinitely

```scala
object ExecutionContextSleep
       extends App {
  for (i<- 0 until 32) execute {
    Thread.sleep(2000)
    log(s"Task␣$i␣completed.")
  }
  Thread.sleep(10000)
}
```

- Expected: all executions terminate after 2s
- Result: only some execute after 2s

- Using quad-core CPU with hyper threading
- `global` has 8 threads in the thread pool

- executes tasks in batches of 8
- after 2s, 8 tasks print "completed"
- after 2s more, 8 more print "completed"
- sleep: all enter a timed waiting state

- if T1 waits for T10 to notify: blocks indefinitely

Lock-free programming

Avoiding `synchronized` with atomic variables

- atomic variable: memory location that supports complex linearizable operations
- ... i.e., appears to occur atomically
- `write` of a volatile operation: simple linearizable operation
- at least two reads and/or writes: complex linearizable operation

Avoiding `synchronized` with atomic variables

- atomic variable: memory location that supports complex linearizable operations
- ... i.e., appears to occur atomically
- `write` of a volatile operation: simple linearizable operation
- at least two reads and/or writes: complex linearizable operation
- java.util.concurrent.atomic supports some complex ones:
    - AtomicBoolean
    - AtomicInteger
    - AtomicLong
    - AtomicReference

Variation of Example 1 (`getUniqueId`)

```
import
    java.util.concurrent.atomic._

object AtomicUid extends App {
  private val uid =
    new AtomicLong(0L)

  def getUniqueId(): Long =
    uid.incrementAndGet()
  execute {
    log(s"Uid␣asynchronously:␣$
        {getUniqueId()}")
  }
  log(s"Got␣a␣unique␣id:␣$
      {getUniqueId()}")
}
```

Compare-And-Set (CAS) – the ♡ of complex linearizable operations

- CAS can be used to implement others:
  - getAndSet
  - decrementAndGet
  - addAndGet
- available in all atomic variables
- including AtomicReference[T]

Long-CAS conceptually equivalent to:

```
def compareAndSet(ov: Long, nv: Long):
              Boolean = this.synchronized {
    if (this.get == ov) false else {
      this.set(nv)
      true
  } }
```

Ref-CAS conceptually equivalent to:

```
def compareAndSet(ov: T, nv: T):
              Boolean = this.synchronized {
  if (this.get eq ov) false else {
    this.set(nv)
    true
} }
```

Using CAS

- Back to Example 1 (getUniqueId)
- Need to keep-on-trying
- Looks like busy-waiting, but it is much better
- Here: using (cheap) recursion instead of a loop

```
@tailrec def getUniqueId(): Long = {
  val oldUid = uid.get
  val newUid = oldUid + 1
  if (uid.compareAndSet(oldUid,
      newUid)) newUid
  else getUniqueId()
}
```

Lock-free programming – really?

- Lock-free programs: without locks (with `synchronized`)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...

Lock-free programming – really?

- Lock-free programs: without locks (with `synchronized`)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...
- (almost):
  - lock-free $\Rightarrow$ use atomic variables (for atomicity)
  - use atomic variables $\not\Rightarrow$ lock-free

Lock-free programming – really?

- Lock-free programs: without locks (with `synchronized`)
- Achieved using atomic variables (and some re-trying)
- No locks, no deadlocks...
- (almost):
  - lock-free $\Rightarrow$ use atomic variables (for atomicity)
  - use atomic variables $\not\Rightarrow$ lock-free

```
object AtomicLock extends App {
  private val lock = new
      AtomicBoolean(false)
  def mySynchronized(body: =>Unit):
      Unit = {
    while (!lock.compareAndSet(false,
        true)) {}
    try body finally lock.set(false)
  }
  var count = 0
  for (i<- 0 until 10) execute {
      mySynchronized { count += 1 } }
  Thread.sleep(1000)
  log(s"Count␣is:␣$count")
}
```

Lock-freedom definition

Lock-freedom

Given a set of threads and an operation OP.

OP is lock-free if at least one thread always completes the operation after a finite number of steps, regardless of the speed at which different threads progress.

One more example: Concurrent filesystem

- **Example 1:** getUniqueId()
- **Example 2:** Logging Bank Transfers
- **Example 3:** Thread pool
- **Example 4:** Batman
- **Example 5:** Concurrent filesystem

Filesystem API

T1 is creating F:

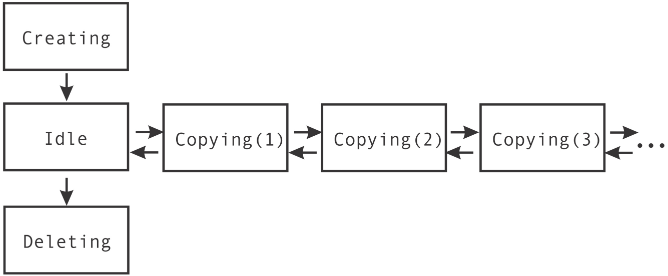
T2 cannot copy or delete F

T1 & T2 are copying F:

T3 cannot delete F

T1 is deleting F:

T2 cannot copy nor delete F



in *"Learning Concurrent Programming in Scala"*, pg. 79

Filesystem API

T1 is creating F:

T2 cannot copy or delete F

T1 & T2 are copying F:

T3 cannot delete F

T1 is deleting F:

T2 cannot copy nor delete F

```
class Entry(val isDir: Boolean) {
  val state = new AtomicReference[State](new Idle)
}

sealed trait State
class Idle extends State
class Creating extends State
class Copying(val n: Int) extends State
class Deleting extends State
```

Deleting and Copying

Deleting: **prepare** (checks for permission) then delete (perform delete)

Copying: aquire (get permission); copy (perform action); then release (give permission)

Prepare for deleting

```
@tailrec
private def prepareForDelete(entry: Entry): Boolean = {
  val  s0 = entry.state.get
  s0 match {
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Deleting)) true
      else prepareForDelete(entry)
    case c: Creating =>
      logMessage("File currently created, cannot delete."); false
    case c: Copying =>
      logMessage("File currently copied, cannot delete."); false
    case d: Deleting =>
      false
  }
}
```

`logMessage`: presented later – similar to `log`, but stores the log message

Bad copy – the ABA problem

"ABA" problem: two readings of the same value A lead to believe that B was never present (type of race condition)

Illustrated by a bad acquire-release for `Copying`, using a mutable `n` in:

```
Copying(var n: Int)
```

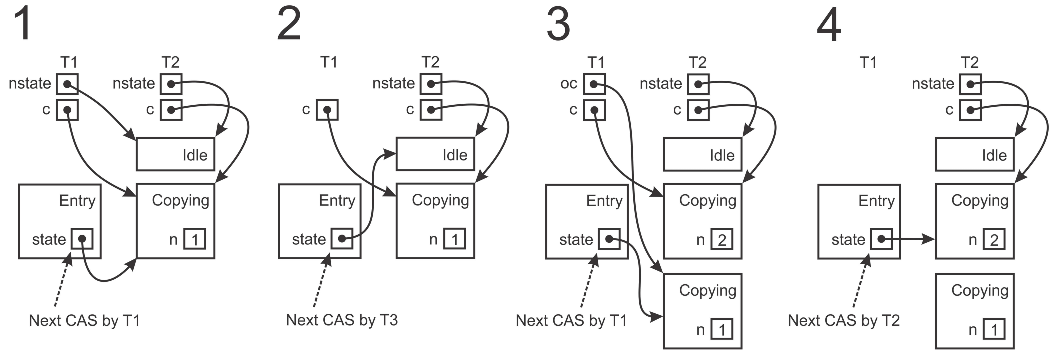
Bad code – acquire/release Copying

```
def releaseCopy(e: Entry): Copying = e.state.get match {
  case c: Copying =>
    val nstate = if (c.n == 1) new Idle else new Copying(c.n - 1)
    if (e.state.compareAndSet(c, nstate)) c
    else releaseCopy(e)
}
```

```
def acquireCopy(e: Entry, c: Copying) = e.state.get match {
  case i: Idle =>
    c.n = 1
    if (!e.state.compareAndSet(i, c)) acquireCopy(e, c)
  case oc: Copying =>
    c.n = oc.n + 1
    if (!e.state.compareAndSet(oc, c)) acquireCopy(e, c)
}
```

Optimization: reusing previous `Copying` if possible

Bad trace – T1&T2 release; T3&T1 aquire – (T2 is slow)



[in *"Learning Concurrent Programming in Scala"*, pg. 82]

Some guidelines to avoid the ABA problem

- use fresh objects in `AtomicReference`
- use immutable objects in `AtomicReference`
- avoid re-assigning the same value to an atomic variable
- only increment or decrement values of numeric atomic variables (if possible)

Lazy values

Lazy values can cause deadlocks

- **lazy values:** initialized when read for the first time
- these should not depend-on/modify state (non-determinism)
- code in singleton objects: lazy execution
- under the hood: first write uses a lock – to ensure at most a thread initialises a lazy value
- stack overflow (sequential code)
  can become
  deadlock (concurrent code)

```
object LazyValsCreate extends App {
  var x = 5
  lazy val y = x+2
  execute {log(s"Wrk: y = $y")}
  x = 10
  log(s"Main: y = $y")
  // y = 7 or 12 in both cases
  Thread.sleep(500)
}
```

```
object LazyValsDeadlock extends App {
  object A { lazy val x: Int = B.y }
  object B { lazy val y: Int = A.x }
  execute { B.y }
  A.x
}
```

Concurrent (mutable) collections

Default mutable collections are not concurrent

- Naive code: arbitrarily returns different results and exceptions
- Corruption of the internal state
- Possible fixes:
  - immutable collections + atomic variables
  - mutable collections + synchronized
  - dedicated libraries

```scala
import scala.collection._
object CollectionsBad extends App {
  val buffer =
      mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) =
      execute {
    buffer ++= numbers
    log(s"buffer␣=␣$buffer")
  }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

Default mutable collections are not concurrent

- Naive code: arbitrarily returns different results and exceptions
- Corruption of the internal state
- Possible fixes:
  - immutable collections + atomic variables
    (does not scale)
  - mutable collections + synchronized
    (assuming collections do not block; may not scale)
  - dedicated libraries
    (far better performance and scalability)

```
import scala.collection._
object CollectionsBad extends App {
  val buffer =
      mutable.ArrayBuffer[Int]()
  def asyncAdd(numbers: Seq[Int]) =
      execute {
    buffer ++= numbers
    log(s"buffer␣=␣$buffer")
  }
  asyncAdd(0 until 10)
  asyncAdd(10 until 20)
  Thread.sleep(500)
}
```

Some concurrent collections

- Concurrent queues
  - java.util.concurrent.BlockingQueue interface
  - ...ArrayBlockingQueue class (bounded)
  - ...LinkedBlockingQueue class (unbounded)
- Concurrent Sets and Maps
  - scala.collection.concurrent.Map trait
  - java.util.concurrent.ConcurrentHashMap class

BlockingQueue API

Operation	Exception	Special value	Timed	Blocking
Dequeue	remove(): T	poll(): T	poll(t: Long, u: TimeUnit): T	take(): T
Enqueue	add(x: T)	offer(x: T): Boolean	offer(x: T, t: Long, u: TimeUnit)	put(x: T)
Inspect	element: T	peek: T	N/A	N/A

[in *"Learning Concurrent Programming in Scala"*, pg. 90]

Back to Example 5: logging in our concurrent filesystem

We will compile a queue of messages when logging messages in our file system

```
class FileSystem(...) {
  ...
  private val messages = new LinkedBlockingQueue[String]
  val logger = new Thread {
    setDaemon(true)
    override def run() = while (true) log(messages.take())
  }
  logger.start()
  def logMessage(msg: String): Unit = messages.offer(msg)
}
```

```
...
val fileSystem = new FileSystem(".") // to be defined later
fileSystem.logMessage("Testing␣log!")
```

Note on iterators

- concurrentQueue.iterator
- can produce inconsistent results
- while traversing and modifying, the iterator can be updated
- (heavier) exceptions create a copy when producing an iterator

Example 5: files as a Map in our FileSystem

```scala
import scala.collection.convert.decorateAsScala._
import java.io.File
import org.apache.commons.io.FileUtils // needs "commons-io" in build.sbt

class FileSystem(val root: String) {
  val rootDir = new File(root)
  val files: concurrent.Map[String, Entry] =
    new ConcurrentHashMap().asScala
  for (f <- FileUtils.iterateFiles(rootDir, null, false).asScala)
    files.put(f.getName, new Entry(false))

  ...
}
```

Deleting a file

Recall the `prepareForDelete(entry)`

```
...
def deleteFile(filename: String): Unit = {
  files.get(filename) match {
    case None =>
      logMessage(s"Path␣'$filename'␣does␣not␣exist!")
    case Some(entry) if entry.isDir =>
      logMessage(s"Path␣'$filename'␣is␣a␣directory!")
    case Some(entry) => execute {
      if (prepareForDelete(entry))
        if (FileUtils.deleteQuietly(new File(filename)))
          files.remove(filename)
    }
  }
}
```

Some complex linearizable methods of concurrent Map

Signature	Description
putIfAbsent (k: K, v: V): Option[V]	**This atomically assigns the value `v` to the key `k` if `k` is not in the map. Otherwise, it returns the value associated with `k`.**
remove (k: K, v: V): Boolean	This atomically removes the key `k` if it is associated to the value equal to `v` and returns `true` if successful.
replace (k: K, v: V): Option[V]	This atomically assigns the value `v` to the key `k` and returns the value previously associated with `k`.
replace (k: K, ov: V, nv: V): Boolean	This atomically assigns the key `k` to the value `nv` if `k` was previously associated with `ov` and returns `true` if successful.

[in *"Learning Concurrent Programming in Scala"*, pg. 95]

- These use "equals" instead of the reference (which CAS does)
- Avoid `null` as key or valye (often used as special values)

- Methods `+=`, `-=`, `put`, `update`, `get`, `apply`, `remove` are (non-complex) linearizable

Wrapping up our Filesystem (Example 5)

Copying in our Filesystem

Recall our broken aquireCopy/releaseCopy methods (ABA problem) – slide19

```
@tailrec
private def acquire(entry: Entry): Boolean = {
  val s0 = entry.state.get
  s0 match {
    case _: Creating | _: Deleting =>
      logMessage("File␣inaccessible,␣cannot␣copy."); false
    case i: Idle =>
      if (entry.state.compareAndSet(s0, new Copying(1))) true
      else acquire(entry)
    case c: Copying =>
      if (entry.state.compareAndSet(s0, new Copying(c.n+1))) true
      else acquire(entry)
  }
}
```

Copying in our Filesystem

Same CAS retry-approach for releasing.

```
@tailrec
private def release(entry: Entry): Unit = {
  val s0 = entry.state.get
  s0 match {
    case c: Creating =>
      if (!entry.state.compareAndSet(s0, new Idle)) release(entry)
    case c: Copying =>
      val nstate = if (c.n == 1) new Idle else new Copying(c.n-1)
      if (!entry.state.compareAndSet(s0, nstate)) release(entry)
  }
}
```

Copying in our Filesystem

Finally: wrapper for copying a file.

```
def copyFile(src: String, dest: String): Unit = {
  files.get(src) match {
    case Some(srcEntry) if !srcEntry.isDir => execute {
      if (acquire(srcEntry)) try {
        val destEntry = new Entry(isDir = false)
        destEntry.state.set(new Creating)
        if (files.putIfAbsent(dest, destEntry) == None) try {
          FileUtils.copyFile(new File(src), new File(dest))
        } finally release(destEntry)
      } finally release(srcEntry)
    }
} }
```

Creating and handling processes

Beyond JVM

- So far: run in a single JVM
- Now: run processes outside JVM
- Why:

Beyond JVM

- So far: run in a single JVM
- Now: run processes outside JVM
- Why:
  - Some programs do not exist in Scala/Java

Beyond JVM

- So far: run in a single JVM
- Now: run processes outside JVM
- Why:
  - Some programs do not exist in Scala/Java
  - Want to sandbox untrusted code

Beyond JVM

- So far: run in a single JVM
- Now: run processes outside JVM
- Why:
  - Some programs do not exist in Scala/Java
  - Want to sandbox untrusted code
  - Performance (running independent code)

Beyond JVM

- So far: run in a single JVM
- Now: run processes outside JVM
- Why:
  - Some programs do not exist in Scala/Java
  - Want to sandbox untrusted code
  - Performance (running independent code)
- Using the `scala.sys.process` package

Using processes - examples

```
import scala.sys.process._
object ProcessRun extends App {
  val command = "ls"
  val exitcode = command.! // run process (with side effects)
  log(s"command␣exited␣with␣status␣$exitcode") }
```

```
def lineCount(filename: String): Int = {
  val output = s"wc␣$filename".!! // run and retreive stdout
  output.trim.split("␣").head.toInt }
```

```
object ProcessAsync extends App {
  val lsProcess = "ls␣-R␣/".run() // run and returns a Process object
  Thread.sleep(1000)
  log("Timeout␣-␣killing␣ls!")
  lsProcess.destroy() } // kill a slow process
```

https://www.scala-lang.org/api/2.13.x/scala/sys/process/ProcessBuilder.html

Wrapping up “concurrency blocks”

- `executor.execute(...)`
- lock-free programming with atomic variables
- `av.compareAndSet(...)`
- ABA problem
- Lazy values & “lazy” objects
- `java.util.concurrent.BlockingQueue`
- `scala.collection.concurrent.Map`
- *weakly consistent iterators*
- *custom concurrent data structures*

- Filesystem example
- Processes outside JVM

Wrapping up "concurrency blocks"

- `executor.execute(...)`
- lock-free programming with atomic variables
- `av.compareAndSet(...)`
- ABA problem
- Lazy values & "lazy" objects
- `java.util.concurrent.BlockingQueue`
- `scala.collection.concurrent.Map`
- *weakly consistent iterators*
- *custom concurrent data structures*

- Filesystem example
- Processes outside JVM

Next

- *Futures and Promises*
- *Data-Parallel Collections*
- *Reactive Programming (Concurrently)*
- *Software Transactional Memory*
- Actors